Chapter 1

Fault Diagnosis of Hybrid Computing Systems Using Chaotic-Map Method

Nageswara S. V. Rao and Bobby Philip

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/intechopen.79978

Abstract

Computing systems are becoming increasingly complex with nodes consisting of a combination of multi-core central processing units (CPUs), many integrated core (MIC) and graphics processing unit (GPU) accelerators. These computing units and their interconnections are subject to different classes of hardware and software faults, which should be detected to support mitigation measures. We present the chaotic-map method that uses the exponential divergence and wide Fourier properties of the trajectories, combined with memory allocations and assignments to diagnose component-level faults in these hybrid computing systems. We propose lightweight codes that utilize highly parallel chaotic-map computations tailored to isolate faults in arithmetic units, memory elements and interconnects. The diagnosis module on a node utilizes pthreads to place chaotic-map threads on CPU and MIC cores, and CUDA C and OpenCL kernels on GPU blocks. We present experimental diagnosis results on five multi-core CPUs; one MIC; and, seven GPUs with typical diagnosis run-times under a minute.

Keywords: fault diagnosis, hybrid systems, chaotic maps, multi-core CPU, GPU

1. Introduction

High performance computing systems utilize increasingly complex *hybrid* nodes that consist of multi-core central processing units (CPU) combined with many integrated core (MIC) or graphics processing unit (GPU) accelerators [1]. The next generation systems that target Exascale computations are expected to be massive with computing elements totaling a million [2, 3]. Furthermore, these computing systems are expected to be built, at least in part, using off-the-shelf components such as CPUs, Accelerated Processing Units (APU) and GPUs, which have an expected life-span in the range of 5–10 years. Consequently, the computations that run

for a few hours on such systems are likely to experience multiple faults, and it is essential to account for them to achieve the *resilience* of these computations [4, 5]. Detection of such faults contributes to resilient computations in a number of ways such as: supporting the quarantine of faulty units from the scheduler pool; replacement of faulty processor boards and accelerators; and, initiation of application migration and check point recovery. However, fast and efficient detection of such faults in hybrid computing systems is complicated due to the continued increases in the number and complexity of processors, accelerators and interconnects.

However, fast and efficient detection of such faults in hybrid computing systems is complicated due to the continued increases in the number and complexity of processors, accelerators and interconnects.

The impact of such faults could be quite significant on certain scientific computations, particularly if they fail to trigger checkpoint recovery or process migration, or result in too many errors that require inordinate number of checkpointing operations. Furthermore, the variety of faults is expected to expand in future as hybrid architectures evolve with increasing numbers of cores, sockets and blocks, and with complex processors and interconnect designs. The component faults in these systems can manifest in a variety of ways: faults in arithmetic and logic units (ALU) and floating point units (FPU) lead to instruction execution errors; and memory element errors and transport errors (over memory bus, inter-processor link, PCI bus connection to GPU memory, or interconnect) lead to erroneous data. In production systems, current support for detecting these faults is somewhat limited, primarily to hardware monitors and codes with "known" outputs, and several other approaches are currently under development [5–18]. In fact, some faults that develop during the computation may not be detected at all, and the computation may indeed run to completion and produce unsuspected erroneous output. One practical diagnosis technique is to run an application and compare its output with *a priori* known correct values. For example, codes such as CUDA-enhanced HPL [19] with known outputs have been used in practice to verify error-free executions. These codes, however, require significant execution times, since they solve dense linear systems with a primary purpose of benchmarking the (error-free) system.

We propose lightweight codes to quickly and efficiently detect component faults in hybrid computing nodes consisting of multi-core CPUs with MIC or GPU accelerators. These codes are based on developing the chaotic-map method[1] to diagnose hybrid systems, and are among the smallest codes capable of detecting a large class of ALU, memory and interconnect errors, typically requiring a few iterations of few instructions. The chaotic-map method is introduced [15] as a fault diagnosis tool for computing systems, and applied to multi-core CPUs using pthreads in [16]; but these codes are not transferable to GPUs due to their significantly different architectures and software support. We first extend the chaotic-map implementation to include logical and integer operations, and develop CUDA and OpenCL kernels to diagnose

[1]Chaotic maps have origins in the analysis of non-linear systems with complex dynamics, such as weather systems. Extensive theory and analysis methods of chaotic maps have been developed [20], and are often used for establishing the existence of chaotic dynamics in a wide range of non-linear system models [21].

Multi-Core CPU:
4-core Intel Xeon 2.67 GHz
16-core AMD Opteron 2.3 GHz
16-core Intel Xeon 2630
32-core Intel Xeon E5-2650 2.7GHz
48-core AMD Opteron 6176 SE 2.29GHz
Single-MIC:
Intel Xeon Phi Coprocessor 3120P/A
Single-GPU:
Quadro 600, Quadro K4200, Tesla T10, Tesla C1060
Tesla K20X, Tesla K20c, AMD Firepro W9000
Single-APU:
AMD A10-7850 K
Multiple-GPU:
8 Nvidia Tesla T10 GPUs
Nvidia Tesla K20c and AMD Firepro W9000
Intel HD Graphics 4000 and Nvidia GeForce GT 650 M

Table 1. Nodes used in implementation and testing of diagnosis codes.

GPUs, and integrate them with pthreads multi-core CPU diagnosis codes to diagnose large systems with hybrid nodes. We have implemented and tested these diagnosis codes on systems shown in **Table 1**, namely on five multi-core CPUs; MIC accelerator; seven GPUs; and, three multi-GPU systems.

Our main objective is to rapidly diagnose the faults in large-scale hybrid computing systems with the architecture shown in **Figure 1**. In particular, we consider detecting component faults entirely by software means, similar in spirit to the approaches of Erez et al. [6] and Sahoo et al. [18]; in particular, we focus on codes that run in a few minutes to diagnose non-transient faults.

A finer diagnosis to pinpoint individual faulty digital gates, as typical in the fault diagnosis literature [22], requires solutions to the underlying NP-hard problems. The general problem of detecting resilience of codes is computationally undecidable in Turing sense [17]. Also, sporadic faults that last for short durations (i.e., micro seconds) are not addressed here. Our diagnosis codes are intended to provide "quick" diagnosis to complement other methods[2] such as hardware monitors, HPL codes [19], application-specific detection methods [23–26], and verification systems [27]. While our original motivation is to support facility operations, our diagnosis codes can be made part of a broader, resilience ecosystem to complement and

[2]Due to the multi-disciplinary nature of the area of extreme-scale resilient computations, the literature on related works is quite extensive, and we only refer to a very small set of works that are directly connected to the technical areas of this report.

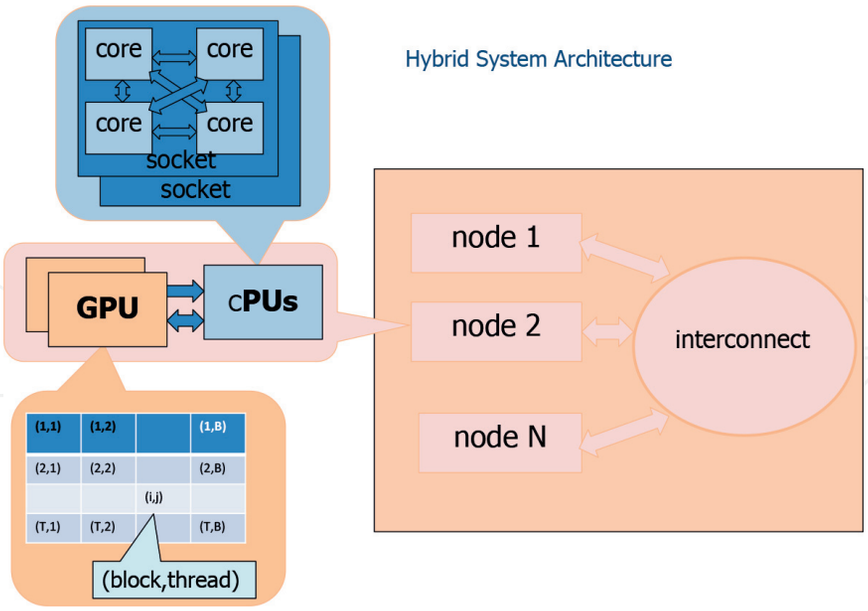

Figure 1. CPU-GPU hybrid system architecture.

support Algorithm-Based Fault Tolerance (ABFT) methods [7, 28]; software-based fault detection [5, 6]; and, likely invariants for detecting hardware faults [18].

Our overall approach is to compute chaotic-map trajectories concurrently on all CPU and MIC cores, and GPU blocks. Two properties of chaotic map trajectories are critical for diagnosis purposes: (a) their *exponential divergence* ensures that the trajectories subjected to faults will rapidly diverge from the majority (fault-free) and are easily detected; and (b) their *density* and *aperiodicity* ensures that they span across and cover a majority of bits involved in the constituent operations. Our codes utilize concurrent threads to compute chaotic map trajectories that are augmented with: (a) arithmetic and logic operations to diagnose ALU operations, and (b) content-preserving data movement operations to diagnose memory elements, busses and interconnects.

This paper is organized as follows. We describe the basics of the fault detection method using chaotic maps in Section 2. We present a brief description of the hybrid architecture and the details of our diagnosis codes in Section 3. We describe the overall diagnosis method in Section 3.1, and provide the details of diagnosis of CPU and MIC cores in Section 3.2, and the details of diagnosis of GPU blocks using CUDA and OpenCL kernels in Sections 3.4 and 3.4, respectively. We present experimental results in Section 4.

2. Diagnosis using chaotic maps

A *Poincare map* $M : \mathfrak{R}^d \mapsto \mathfrak{R}^d$ specifies a sequence, called the trajectory, of a real-vector *state* $X_i \in \mathfrak{R}^d$ that is updated at each iteration i such that $X_{i+1} = M(X_i)$ [21]. The computation of $M(X_i)$ may involve floating-point operations, such as multiplication and addition, and logical

operations such as comparison of numbers, and can vary significantly in the number and types of operations. The trajectory $X_0, X_1, \ldots, X_t$, such that $X_i = M^i(X_0)$, generated by certain Poincare maps can exhibit complex profiles, even when M is computationally simple. In **Figure 2**, we show trajectories of the *logistic map* $M_{L_a}(X) = aX(1-X)$, for $X \in [0,1]$, which requires two multiplications and one subtraction per iteration. In **Figure 3**, we show the trajectories of the *tent map*

$$M_{T_b}(X) = \begin{cases} bX & \text{if } X \leq 1/2 \\ b(1-X) & \text{if } X > 1/2 \end{cases}$$

for $X \in [0,1]$, which requires a comparison operation, one multiplication and at most one subtraction per iteration. The trajectories of these maps exhibit complex dynamics as shown in **Figures 2(a)** and **3(a)** for the logistic map for $a = 4$ and the tent map for $b = 2$, respectively. The trajectories generated by the Poincare map M are characterized by the *Lyapunov exponent* defined as $\mathcal{L}_M = \ln\left|\frac{dM}{dX}\right|$, which characterizes the separation of the trajectories that originate from the nearby states. For example, the Lyapunov exponent of the tent map is $\mathcal{L}_{M_{T_b}} = \ln b$, which is defined for all $X \in [0,1]$ except at $X = 1/2$.

A bounded trajectory $X_0, X_1, \ldots$ generated by the Poincare map $M(.)$ is *chaotic* if (i) it is not asymptotically periodic, and (ii) Lyapunov exponent $\mathcal{L}_M$ is greater than zero [21]. Two important properties of the chaotic maps are exploited here for fault diagnosis: (a) the *exponential divergence* ensures that trajectories whose states slightly differ from each other at any iteration

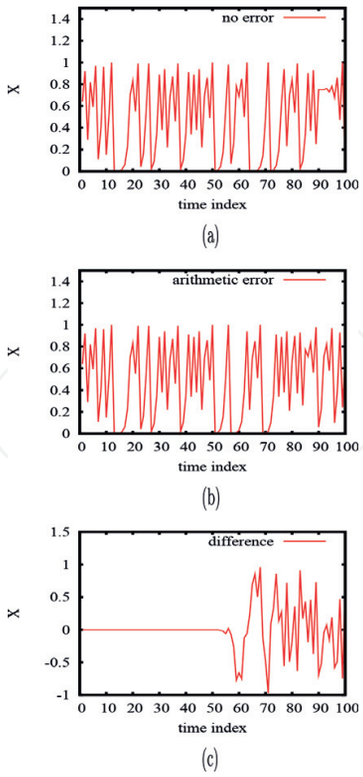

Figure 2. Trajectories of the logistic map. (a) trajectory with no errors, (b) trajectory under arithmetic error, (c) differences in trajectories with and without error.

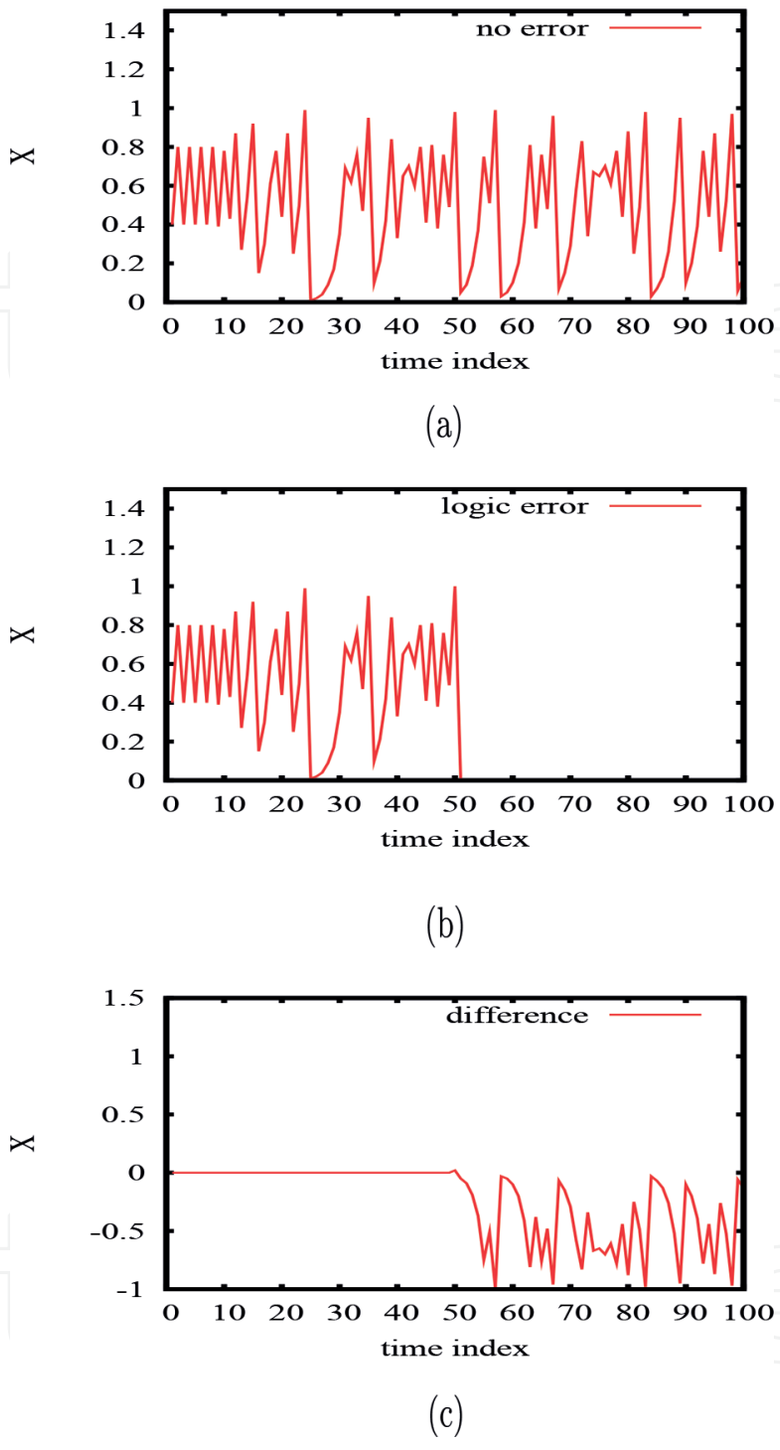

Figure 3. Trajectories of the tent map. (a) trajectory with no error, (b) trajectory with logic error, (c) difference in trajectories with and without error.

rapidly diverge within a few steps, and (b) the *high spatial density* and *broad Fourier spectrum* of states of a trajectory spreads them across the bit-space of the underlying computing operations within a few iterations. The first property has been proposed in [15] as an effective computational mechanism to rapidly amplify errors caused by factors such as bit flip in memory content or stuck-at fault in an ALU operation. We extend this approach to diagnose High

http://dx.doi.org/10.5772/intechopen.79978

Performance Computing (HPC) systems with small fault rates: these maps are computed concurrently so that errors are detected by comparing them to a small majority of them. In addition, several chaotic maps have very small computational requirements, and a vast literature is available on the analytical [8, 20], statistical [29] and computational aspects of these maps [13, 30]. It is possible in theory to utilize linear maps in a similar way, but they are not as efficient in detecting "small" errors (such as in least significant digits) which have to be linearly amplified through multiple iterations to trigger detection, and also they do not generate dense states and hence are limited in their bit-level coverage.

The trajectories that slightly differ from each other in any iteration rapidly diverge from each other in a few steps, as shown in **Figure 2(a)** and **(b)** for the logistic map, and **Figure 3(a)** and **(b)** for the tent map. This property is utilized as a mechanism to rapidly amplify errors in computations caused by factors such as bit flip in memory elements or stuck-at fault in an ALU operation. Also, through the iterations, the states are spread across the interval $[0, 1]$ so that the bit-space of the underlying computations, for example, of the registers, is covered with a high likelihood. The difference between two trajectories with the same starting state is shown in **Figure 2(c)** for the logistic map, where the state is corrupted by 1/10000 magnitude in iteration 50. During iterations 0 through 50, the difference between the trajectories is 0, but the small difference in state magnitude is amplified to above 0.25 within 8 iterations, which is typically under 1 ms on the systems we tested. In **Figure 3**, we show the effect of error in the logical operation, wherein the result of the comparison is flipped in iteration 50. The effect on the trajectory is more dramatic as shown in **Figure 3(b)**, and the difference in the trajectories crosses 0.25 within two iterations. Such divergence in trajectories can be detected by a magnitude test, and the detection time is controlled by the Lyapunov exponent of the map. While both logistic and tent maps provide exponential divergence, they cover the state space [0, 1] differently as illustrated in **Figure 4**, as a result of different Lyapunov exponents. It is lower in the middle and large at the ends of state space [0, 1] for the logistic map, but is uniform for the tent map, which makes it preferable for its coverage to support diagnosis.

The computation of a chaotic map $M(.)$ is sensitive to errors in its constituent operations, and the mechanisms used in storing and updating the states. The detectable faults include errors in arithmetic and logical operations performed by ALU, and faults in registers and memory, but

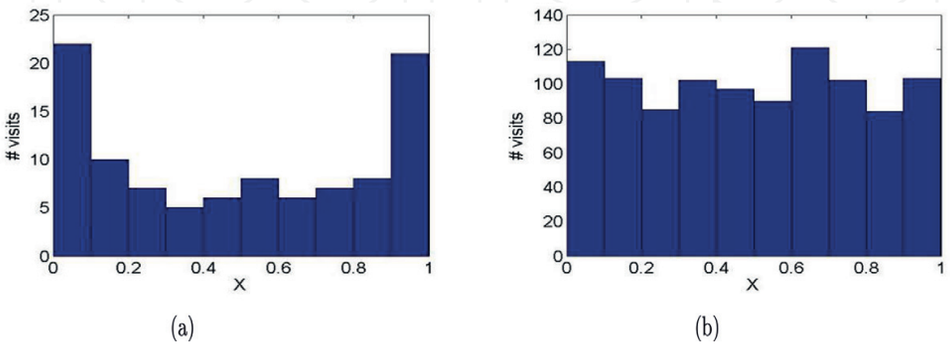

Figure 4. State space coverage by chaotic trajectories. (a) coverage under logistic map, (b) coverage under tent map.

are limited to the basic operations of $M(.)$ itself. We propose the *chaotic-detection map* $M_D(.)$[3] that augments the detection space of $M(.)$ so that its execution path is routed through different computing operations, memory locations and interconnect links to capture errors due to them. The chaotic-detection map is obtained by adding the following two types of operations to the chaotic map iterations.

a. **Augmenting computing operations:** The chaotic map iterations are augmented with sequences of logical and arithmetic operations, which are selected based on the instruction sets of CPU cores and GPU blocks to complement the original chaotic map operations. The same sequence of operations is used in all concurrently computed maps by using the same process code for pthreads of all CPU cores and the same CUDA kernel on all GPU blocks. The type of state variable is used to exercise different parts of the computing units; in particular, it is scaled to a large integer and addition and multiplication are applied to exercise integer operations. Also, special operations such as log are applied to X_i to exercise special instructions that are implemented in hardware by Extended Math Unit (EMU), when applicable. The type casting of variables is used to exercise single precision and double precision processing units as well as Vector Processing Units (VPU), as described in the next section.

b. **Content-preserving data movement operations:** The state variable X_i is moved among the memory elements and/or across the interconnects in between applying $M(.)$ iterations, to capture errors in the memory elements and paths, and during the transmission across the interconnects. In each operation, the contents of X_i are unchanged under failure-free conditions. These movements can be realized by several means based on the supported operations, ranging from simple assignment statements to employing additional variables in the "shared" memory to utilizing explicit MPI, CUDA or other constructs. In particular for multi-core processors, memory assignments using pthreads can be used for both purposes, namely, to test memory unit errors as well as transport errors across the memory bus or hypertransport.

The rate of divergence of a chaotic map, and hence the detection times of failures depend on the Lyapunov exponent $\mathcal{L}_M$, generally larger values leading to quicker divergence. The class of faults detected by a chaotic-detection map depends on the chaotic map and the augmenting and data movement operations as well as the computing units used for their computation. A main consideration in developing the diagnosis codes is to efficiently compute the chaotic-detection maps on computing units with identical parameters, sequences of augmenting operations, state-preserving movement operations and chaotic map updates, so that the end states are identical under fault-free conditions. Their implementation critically depends on the software primitives supported on the systems, and they in turn are closely tied to the underlying system architecture, including the location of the computing elements, memory hierarchies and interconnects. In the next section, we describe specific implementations to compute chaotic-detection maps on multi-core CPUs, MICs, GPUs and hybrid systems.

[3]The chaotic-detection map is a generalization of the chaotic-identity map proposed in [15], which was restricted to the operations with inverses.

http://dx.doi.org/10.5772/intechopen.79978

3. Hybrid system diagnosis

We consider hybrid computing systems, wherein each node consists of muti-core, possibly multi-sockets CPUs, and one or more GPU and XeonPhi MIC accelerators; in the limiting case, we have a single node with a multi-core CPU and zero or more GPUs. Computations are spawned to run on CPU cores using OpenMP, pthreads or similar constructs, and on GPU blocks using CUDA or OpenCL threads. Within each node, however, the data movements are carried out differently on CPUs and GPUs, since the former accesses different levels of on-board memory, but the latter can only directly access memory physically located on the GPU. The CPU-GPU data transfers are realized using CUDA or OpenCL in our case using memory copy operations between CPU on-board memory and GPU device memory. Computations on GPUs utilize thread bundles on GPU blocks using kernels written in CUDA C or OpenCL. Kernels are launched from the CPU of a node onto the corresponding GPU blocks as a collection of threads. In this section, we describe different components of the diagnosis codes for hybrid systems based on the chaotic-detection maps. Since these can be used as stand-alone codes for simpler systems, codes for a single-node with multi-core CPU with zero or more GPU accelerators, or as a cluster with only CPUs, are presented in separate sections.

3.1. Node diagnosis module

The overall diagnosis strategy is to utilize the "reliable" nodes to launch a *node diagnosis module* on each node as shown in **Figure 5**, under the working assumption that only a small number of

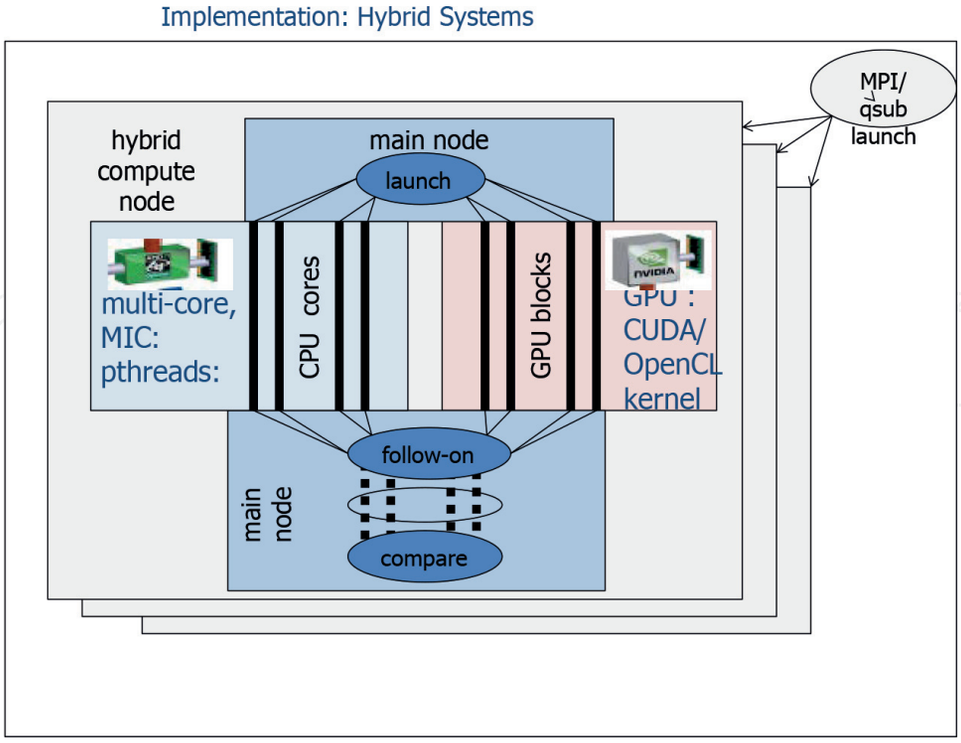

Figure 5. Diagnosis approach for hybrid computing systems.

nodes contain faulty components and a majority of them are fault-free. The node diagnosis module is written in C, and consists of a CUDA or OpenCL C kernel for GPUs, and pthreads code for multi-core CPUs or MIC accelerators. On each node, the diagnosis module detects the number of cores (more precisely, the processing units) using linux/proc system, and also explicitly checks for the physical presence of GPUs using CUDA C or OpenCL system calls (to avoid GPU emulations). It then allocates and initializes node-level global memory and copies the contents onto on-board device memory of GPUs connected to the node. Next, from the default CPU core, it launches concurrent threads to compute the chaotic-detection maps on the computing units, namely CPU cores and GPU blocks, and gathers their outputs and generates the diagnosis output.

The chaotic-detection map computation carried out by threads on each CPU core and GPU block consists of the following basic steps, which will be customized to CPU and GPU architectures (as described in the next sections):

i. Local memory is allocated and initialized based on the specifics of CPU or GPU;

ii. Initial state X_0 of the chaotic-detection map, and additive and multiplicative factors, denoted by A and M respectively, the numbers of pre and post iterations N_1 and N_2, respectively, are accessed so that all threads use the same values;

iii. N_1 iterations of the chaotic-detection map are computed using starting state X_0 to obtain X_K. followed by a sequence of augmenting operations, for example, addition and subtraction of the additive factor A, and division and multiplication with factor M, namely,

$$X_K = X_K + A;$$

$$X_K = X_K - A;$$

$$X_K = X_K/M;$$

$$X_K = X_K * M,$$

to check addition, subtraction, multiplication and division operations.

iv. A fixed sequence of content-preserving data movement operations are performed on variable X_K that are specific to CPU or GPU; and

v. N_2 iterations of the chaotic-detection map are computed with starting state X_K to obtain final state X_E.

At the completion of threads, outputs X_E's from all concurrent threads are transferred back to the default CPU core and are used as starting states for N_3 iterations of a follow-on chaotic map. This follow-on chaotic map amplifies the errors captured by the outputs of chaotic-detection maps from the CPU and GPU threads as well as those occur during data transfers, for example, from GPU to CPU over the PCI bus.

The final outputs X_F's of the follow-on chaotic map are compared to a pre-computed correct state (or to the majority of a small subset of them). If X_F's of all the threads match then no error is declared. If not, diagnosis steps (iii)–(iv) are executed separately to identify portions during

which errors occurred. The step (iii) identifies ALU errors in executing +, −, / and * operations, and other operations of interest can be added. Steps (i), (ii) and (iv) are customized to match the memory architectures of CPU and GPU as described in the following sections, wherein assignment operations are used to diagnose memory elements and data paths. The memory and interconnect diagnosis codes described here, however, are limited mainly to illustrate the detection of faults in memory elements and data paths rather than complete sweeps of memory and interconnects.

All threads that compute chaotic-detection maps are provided identical parameters in step (ii). These parameters are setup on global arrays $GM[.,.]$ of size S_G on each node, which are accessible to all CPU cores and are explicitly transferred to GPU memory. For initialization on the node, malloc call is used for allocating the memory and memset is used to fill the memory with the values. The data movements in step (iv) do not alter the contents if there are no errors in storage or transfer operations, but are designed specifically to match the memory architecture of CPU and GPU; in particular, primitives such as assignments can be used to diagnose memory and transfer errors as will be described in next sections.

3.2. Multi-core CPU diagnosis

Multi-core CPU systems are composed of one or more sockets, each housing a number of processor cores connected to memory modules, which are typically organized in a hierarchy. An example of a single socket quad-core system is HP Z400 workstation shown in **Figure 6** consisting of four 2.67 GHz Intel Xeon CPUs. The cache memory units are connected over the memory bus such that L1 caches are local to processor cores, L2 caches are shared between pairs and L3 caches are global. The memory caches are connected over a combination of memory bus and hypertransport links. The L1 cache is local to cores whereas global memory is accessible to certain cores via hypertransport links. Thus, certain memory transfers between local and global memory take place over hypertransport links.

We now provide the details of node diagnosis module that is specific to multi-core CPUs. It partitions the global memory into non-overlapping parts assigned to N_C processor cores, and launches dedicated threads one on each core as shown in **Figure 7**. Processor core i is assigned the subarray $GM[i,.]$ of size S_{G_i}. Then, a single thread is invoked on each core i using pthread_setaffinity_np call, and this thread computes the chaotic-detection map with the following expanded steps described in Section 3.1.

(i) Local memory is allocated and initialized within the thread as an array $LS[.]$ of specified size S_L using malloc and memset.

(iv-a) The variable X_K is stored and retrieved from each element of the initialized local memory:

for $j = 1, \ldots, S_L$.

$LS[j] = X_K;$

$X_K = LS[j];$

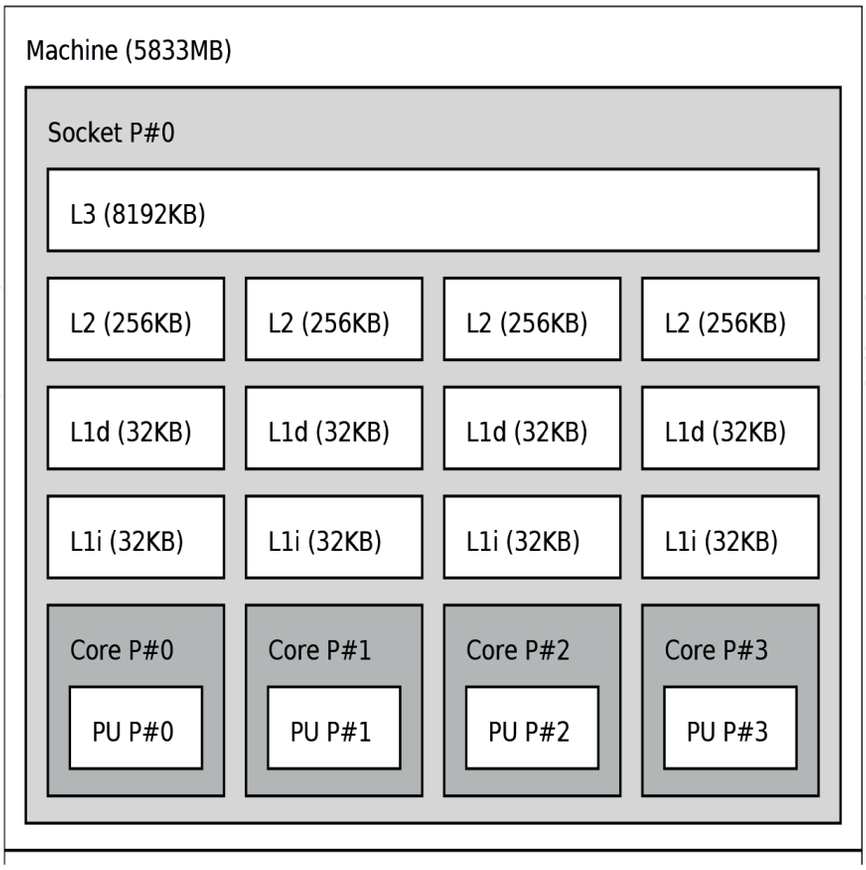

Figure 6. Architecture of 4-core HP Z400 workstation.

These operations are carried out between core i and its local memory.

(iv-b) The variable X_K is stored and retrieved from each element of the assigned partition of the global memory such that the thread assigned to processor code i executes the following code.

for $j = 1, \ldots, S_{G_i}$.

$$GM[i, j] = X_K;$$

$$X_K = GM[i, j];$$

The basic idea of steps (iv-a) and (iv-b) is to utilize the variable assignments to diagnose both memory elements as well as data paths. The step (iv-a) exercises the local memory operations and detects errors in the memory elements, as well as during transport by the memory controller. The step (iv-b) exercises the processor interconnect, as well as the elements in global memory; the interconnect is memory bus for HP z400 workstation, and hypertransport for HP Proliant server. While these steps do not cover all possible errors, they are likely to capture several major errors in ALU, memory and interconnect so that processors with detected errors can be excluded from computations or their boards may be replaced. This memory diagnosis part can be further refined: (a) NUMA tools can be utilized to explicitly allocate memory in different locations and layers so that the assignment operations require data to be transferred across the memory connections, and (b) assignment primitives under higher level constructs

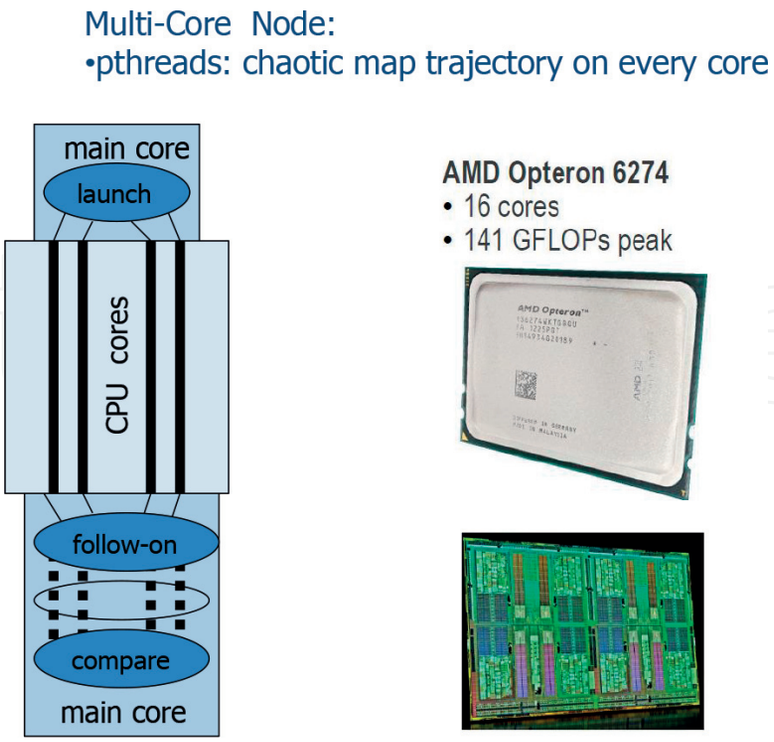

Figure 7. Diagnosis of CPU cores.

such as openMP, SHMEM or PGAS can be used as diagnose memory paths at much higher level using fairly simple codes.

3.3. Xeon phi diagnosis

Intel's first generation MIC Architecture, code named Knights Corner has 57–61 cores per coprocessor. The particular coprocessor we had access to was the Intel Xeon Phi 3120P/A coprocessor with 57 cores, 1TF double precision performance, 6GB GDDR memory with 240 GB/s data transfer rates and a 1.1 GHz clock. Internally the 3120P consists of 57 cores with each core having a VPU, a x87 math co-processor providing double precision transcendentals (non-vectorized) and a scalar processing unit. The VPU on each core internally consisted of 8 double precision FPUs and 16 single precision (SP) FPUs and an extended math unit (EMU) providing single precision vectorized transcendental functions. The VPUs are capable of 8 DP or 16 SP operations per clock cycle. Associated with each VPU are four hardware thread execution contexts each having access to 32,512 bit wide private registers (zmm0-zmm31)and 8 16 bit mask registers, 7 of which are writeable. Associated with each core are 32 KB L1 data and instruction caches and a unified 512 KB instruction and data L2 cache. The instruction set associated with the VPUs is the Intel Initial Many Core Instruction (IMCI) set. The 57 cores are on a round robin bidirectional ring interconnect with 8 memory controllers. For further details we refer to [14, 31–33].

The Xeon Phi node diagnosis module tests the single and double precision ALUs within each VPU, the x87 math coprocessor for each core, the EMU for each core, the general purpose

vector registers for each hardware thread, and the L1 and L2 caches. We did not attempt to design explicit tests for the memory controllers, RAM, or the interconnects to the CPU within this module though in theory this should also be possible. The code is written in C and IMCI assembly. The icc compiler was used to automatically generate vectorized code for the VPUs. The code can be executed either by offloading to the coprocessor or by natively executing on the coprocessor. pthreads is used to spawn off one thread for each logical core detected. For the Intel Xeon Phi 3120P/A coprocessor that we had access to this worked out to a total of 57×4 threads corresponding to the 4 hardware threads associated with the 57 cores. The core affinity for each thread was explicitly set using the **pthread_attr_setaffinity_np** call. Each thread executes a set of chaotic-map detection routines based on the description in Section 3.1 customized to test the hardware components listed above. In particular, it was necessary to write both single and double precision chaotic map routines that operated on 64 bit aligned arrays in order to exercise the SP and DP ALUs. Testing of the $\times 87$ math co-processor was done through the compiler switches **-mmic -fp-model strict** that disabled auto-vectorization and forced the $\times 87$ coprocessor to be exercised. This was verified through examining the generated assembly code. Diagnosis of the EMU was ensured by introducing transcendental function calls. From examining the generated IMCI assembly it was clear that the generated code did not exercise all 32 of 512-bit vector registers (zmm0–zmm31) and mask registers associated with each thread. In order to test the registers, assembly routines were written to span all registers associated with each thread that performed the chaotic map iterations.

3.4. GPU diagnosis using CUDA kernels

Nvidia general purpose GPUs (GPGPU) can be viewed as a set of streaming multiprocessors (SMs) [34] as shown in **Figure 8**. Each SM internally consists of a number of simplified cores: CUDA cores which consist of scalar SP floating point and arithmetic ALUs, DP cores, Special Function Units (SFU) for transcendental functions, and Load/Store units. The number of cores and the relative ratio of the different types has varied from generation to generation. Each SM has a number of schedulers and instruction dispatch units associated with it, as well as a register file shared by all cores in the SM, and local memory partitioned as shared memory and L1 cache. SMs also have access to global device memory. The basic scheduling unit for Nvidia GPGPUs is a warp which consists of 32 threads which operate in SIMT (single instruction multiple thread) fashion. At a higher level, threads are organized into thread blocks and on each block all threads execute concurrently as shown in **Figure 9**. These computations have only direct access to memory on the device DRAM with support from caches. **Table 2** lists hardware specifications for some of the Nvidia microarchitecture generations [9–12]. We note that we have tested on a variety of Nvidia GPGPUs including Quadro 600, Tesla C1060, Quadro K5000, and Tesla K20X (**Table 2**).

The diagnosis module for Nvidia GPUs performs separate chaotic-map detection iterations to detect faults on the CUDA cores, the DP cores, and the SFU components. The global memory $GM[.,.]$ is copied by the node diagnosis module onto the device memory $GM_G[.,.]$ to make it accessible to GPU threads. The thread computations are implemented by a CUDA kernel that is loaded and executed on GPU(s). The same kernel code is executed on each block, which

http://dx.doi.org/10.5772/intechopen.79978

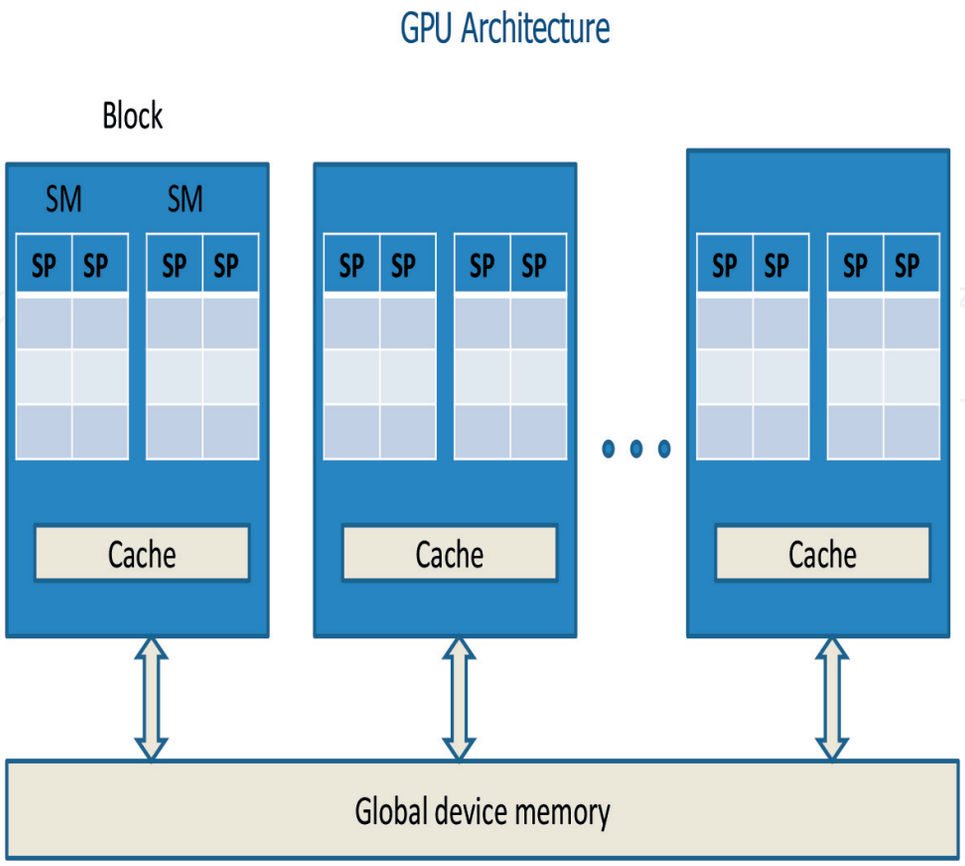

Figure 8. Architecture of a CUDA-capable GPU.

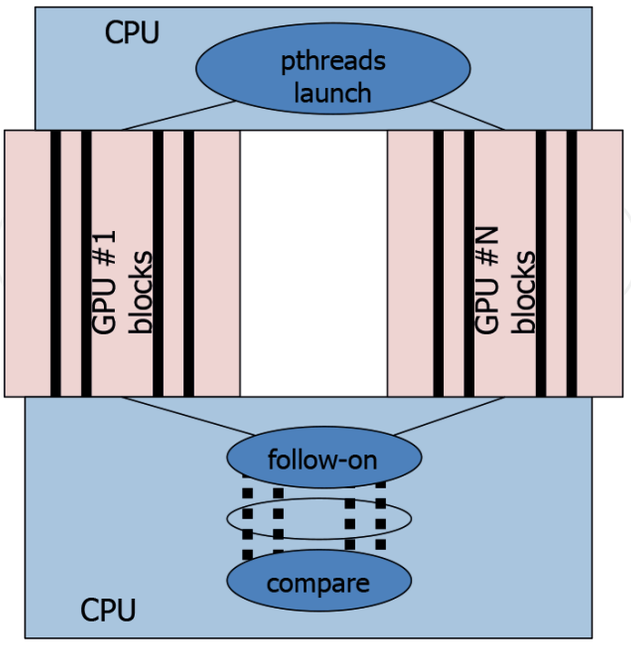

Figure 9. Diagnosis of GPU blocks.

	Tesla	Fermi	Kepler	Maxwell
SM	30	16	8	16
CUDA cores/SM	8	32	32	192
DP cores/SM	1	—	64	4
SFUs/SM	1	4	32	32
Load/Store/SM	—	16	32	32

Table 2. Nvidia GPGPU micro-architecture specifications.

consists of a number of threads that compute the chaotic-detection maps. The chaotic-detector maps computed by each GPU thread consist of the following expanded versions of the steps described in Section 3.1.

(i) The GPU SM id, thread id, warp id, and warp lane numbers are obtained, and the global device memory is allocated to thread T as an array $GM_{G:T}[.]$ of specified size S_T.

(iv-a) A set of variables X_K corresponding to the FP and integer portions of the CUDA core, DP core, and SFU are retrieved from each thread.

(iv-b) The variables X_K are stored and retrieved from each element of the assigned partition of the global memory such that the thread T executes the following code.

for $j = 1, \ldots, S_T$.

$$GM_{G:T}[i,j] = X_K;$$

$$X_K = GM_{G:T}[i,j];$$

(v) N_2 iterations of the chaotic-detection map are computed with starting state X_K to obtain final state X_E;

The GPU diagnosis code is written as a CUDA kernel which is launched from CPU onto GPU blocks concurrently with pthreads on CPU cores. This diagnosis code can be significantly simplified for simpler diagnosis tasks, for example, checking PCI bus transfer errors between CPU and GPU, by simply writing and reading back the global memory arrays.

3.5. GPU diagnosis using OpenCL kernels

AMD GPGPUs and APUs have transitioned to the graphics core now (GCN) Architecture with the GPGPUs and APUs containing multiple GCN compute units (CU). For example, the AMD Radeon HD7970 (Firepro W9000) architecture consists of 32 GCN CU's operating with a 975 MHz clock while the AMD A10-7850 K Kaveri APU architecture consists of 2 Steamroller CPU cores (3.7–4 GHz) and 8 GCN CUs (720 MHz) with a unified address space of up to 32GB. Each CU consists of 4 Vector Units (VU) and one Scalar Unit. Each VU in turn consists of 16 SIMD multi-precision ALUs and a register file. In the case of the Firepro W9000 this is a total of 2048 ALUs (32CUs $\times$ 4VUs $\times$ 16 ALUs) with a theoretical peak of up to 1 TF DP and 4 TF SP performance. The basic scheduling unit of work for a GCN CU consists of a wavefront which is 64 threads. Each

SIMD VU within a CU has its own program counter and instruction buffer which can contain up to 10 wavefront buffers. Four CUs currently share a 32 KB L1 instruction cache. At a given cycle the 4 VUs of one CU can be operating on different wavefronts with a given wavefront completing in 4 clock cycles. In effect, a single CU could have up to 4×10 wavefronts in flight. Associated with each GCN CU is a general purpose register file which consists of 4 independent slices, one for each VU. Each slice consists of 256 vector registers (vGPRs) shared across 10 wavefronts with each vGPR being 64 lanes of 32 bits allowing 64 SP or 32 DP values to be stored in each vector register at each cycle. For further details we refer to [35].

The diagnosis module for AMD GPUs is written in OpenCL with separate kernels for chaotic map fault detection of SP and DP FPUs, integer ALUS, extended math units, and register files. Since the diagnosis modules are written in OpenCL they also run on multi-core CPUs and GPUs from other vendors such as Nvidia. However, obtaining thread level information is much harder with OpenCL and requires the use of vendor specific analysis tools. Hence, the OpenCL chaotic map detection modules provide less quantitative information than the modules written with pthreads for Xeon Phi and CUDA for Nvidia GPGPUs. However, these diagnostic codes provide a good tool for fault detection on hybrid systems due to the cross-platform portabiity that OpenCL provides.

4. Experimental results

The diagnosis codes have been implemented in C using *float* and *double* datatypes based on the logistic and tent maps, and have been developed and tested in stages on the systems listed in **Table 1**. The *test modes* are represented as multi-core CPU (C), manycore processor (MC), single GPU (G), multiple GPUs (MG) or hybrid node with CPU and GPU (CG). The tests were carried out in C, MC, G, MG, and CG modes. In these systems, Xeon Phi's and GPUs are attached to CPUs over PCI bus, and are used as accelerators in all our systems, except in HP Z200 workstation where the GPU is used only for display. The following are the details of systems used in our code implementation and testing.

(C) **Multi-Core CPU:** Five different multi-core systems: 4-core Intel Xeon 2.67 GHz, 16-core AMD Opteron 2.3 GHz, 16-core Intel Xeon, 32-core Intel Xeon 2.7GHz, and 48-core AMD Opteron 2.29GHz.

(MC) **MIC Processors:** Intel Xeon Phi 3120P/A coprocessor.

(G) **Single-GPU:** Six cases: Quadro 600, Quadro K5000, Tesla T10, Tesla C1060, Tesla K20X, and AMD Firepro 9000 GPUs connected to CPU over PCI bus.

(MG) **Multiple-GPUs:** Two cases: 4-socket 48-core HP server with eight 8 Tesla T10 GPUs connected over four PCI busses and Apple MacBook Pro with an Intel HD Graphics 4000 GPU and a Nvidia GT 650 M GPU.

(CG) **Hybrid nodes:** AMD A10-7850 K Kaveri APU with 2 Steamroller CPU cores (3.7–4 GHz) and 8 GCN CUs (72 MHz)

Together these systems represent quite different software environments and architectures, and our diagnosis codes are compiled separately on them under Linux-like environments. But otherwise these codes are portable using C, CUDA, and OpenCL compilers with the pthreads libraries. The diagnosis test results are qualitatively quite similar across these systems. So, we present high-level summaries of our results with some representative traces, and briefly describe (somewhat) unexpected cases that may require additional considerations for developing application codes. For multi-core CPUs no errors were detected on these systems, and as expected the chaotic map outputs X_E's are identical for all cores. We simulated different faults to verify the functionality of diagnosis codes. When GPU are utilized, some interesting precision and emulation artifacts were observed in G and MG modes (Section 4.3).

4.1. Multi-core CPU diagnosis

Four different CPUs have been tested in C mode. The outputs X_E's of all chaotic-detection maps are identical in all these systems, and the results for the 4-core CPU are shown below using the tent and logistic maps ($N_1 = 20, N_2 = 0$ iterations with $X_0 = 0.2$):

Tent map:

Core 0: output: 0.165669 : 3E29A528

Core 1: output: 0.165669 : 3E29A528

Core 2: output: 0.165669 : 3E29A528

Core 3: output: 0.165669 : 3E29A528

Core 0: output: 0.165669 : follow_on: 0.919737

Core 1: output: 0.165669 : follow_on: 0.919737

Core 2: output: 0.165669 : follow_on: 0.919737

Core 3: output: 0.165669 : follow_on: 0.919737

Logistic map:

Core 0: output: 0.787269 : 3F498A78

Core 1: output: 0.787269 : 3F498A78

Core 2: output: 0.787269 : 3F498A78

Core 3: output: 0.787269 : 3F498A78

Core 0: output: 0.787269 : follow_on: 0.062074

Core 1: output: 0.787269 : follow_on: 0.062074

Core 2: output: 0.787269 : follow_on: 0.062074

Core 3: output: 0.787269 : follow_on: 0.062074

http://dx.doi.org/10.5772/intechopen.79978

Outputs from all four threads from the individual cores are identical indicating no errors. The output consists of two parts: first part shows the chaotic-detection map outputs X_E's from the individual cores, and the second part shows the outputs of follow-on chaotic map X_F's. In the first part, the state of chaotic-detection map X_E is printed in C float format in the first column, and in hexadecimal representation in the second column. For the follow-on chaotic map, X_E and X_F are shown in the first and second columns, respectively. These outputs are the same in all four multi-core CPUs and three systems tested.

Since there are no errors detected on the CPU cores above, we simulated four types of errors:

a. We add a small quantity to X_K during the arithmetic operations for thread 0 to simulate ALU errors.

b. We simulate stuck-at memory errors by clamping X_K to a fixed value 0.000001 during the store and retrieve operation for thread 1.

c. We simulate data path errors by replacing X_K by a randomly generated number for a thread 2 during the store and retrieve operation.

d. We flip the outcome of the logical operation in one iteration in X_K computation for thread 3.

The faults (a)–(c) are applicable to both logistic and tent maps, and fault (d) is applicable only to the tent map.

The output for 4-core processor with four faults simulated on different cores, namely type (a) through (d) on cores 0 through 3, respectively, are shown below for the tent map:

Diagnosis summary:

Core 0: output: 0.000370 : 39C21000

Core 1: output: 0.000001 : 358637BD

Core 2: output: 0.010000 : 3C23D70A

Core 3: output: 0.000510 : 3A05A000

Core 0: output: 0.000370 : follow_on: 0.117106

Core 1: output: 0.000001 : follow_on: 0.960860

Core 2: output: 0.010000 : follow_on: 0.795249

Core 3: output: 0.000510 : follow_on: 0.045228

The outputs from these threads are different from those above indicating an error during the execution of each of them. Additionally, the final outputs of each of these chaotic trajectories are different from each other indicating different types of faults. The global memory $GM[.,,]$ is allocated prior to invoking the threads on the processors cores, and local memory $LS[.]$ is allocated within the thread assigned to a particular core. Since memory movements are carried out by all cores between their local memory and global memory, significant portion of the

memory data paths are exercised by the diagnosis code so that major errors in memory bus and interconnect can be detected. An exhaustive coverage of all memory data paths would require extensions of this method such as explicit placement of processes and their memory near the cores, which may be achieved using NUMA tools.

4.2. Xeon phi diagnosis

Below we show a partial output from running the chaotic tent map detection on the Xeon Phi. For lack of space we have only shown the outputs for a few cores and for the SP and DP ALU calculations. Similar fields are outputted for the EMU, the integer ALU the $\times 87$ math coprocessor, and the vector registers for each hardware thread. In addition to the partial summary shown a detailed output is written to file.

Diagnosis summary:

Number of cores detected = 228

Core 003: SP-ALU : 3F7E86A2, DP-ALU : 3FD711AFCA21B76F

Core 000: SP-ALU : 3F7E86A2, DP-ALU : 3FD711AFCA21B76F

Core 001: SP-ALU : 3F7E86A2, DP-ALU : 3FD711AFCA21B76F

Core 002: SP-ALU : 3F7E86A2, DP-ALU : 3FD711AFCA21B76F

…

Core 226: SP-ALU : 3F7E86A2, DP-ALU : 3FD711AFCA21B76F

Core 227: SP-ALU : 3F7E86A2, DP-ALU : 3FD711AFCA21B76F

Outputs from the different threads are identical indicating no errors for the various components tested.

4.3. GPU diagnosis

Four GPUs have been tested in G mode and one is tested in MG mode with pthreads. A single thread is used on each block to compute the chaotic-detection map. The outputs of chaotic-detection maps are identical in all these cases, analogous to the CPU case, when the chaotic-map output is computed without adding the index value to X_F; also, the results are the same as the CPU case when faults were simulated. Recall that to keep track of outputs from individual blocks, the index (block number) was added to X_E, which was then subtracted on CPU to compute X_F. This specific combination of operations involving integers and fractions yielded non-uniform precisions among different blocks of the same GPU. We now briefly describe the details of such cases, and such effects have been observed on all GPUs in **Table 1**. The outputs of diagnosis codes from GPU of Titan using the logistic map are shown below in a condensed form so that only lines corresponding to blocks with different outputs are shown (when no faults are simulated):

http://dx.doi.org/10.5772/intechopen.79978

Chaotic detection map:

block_x[0] = 0.682320 <-> 3F2EAC8E

…

block_x[2] = 2.682321 <-> 3F2EAC90

…

block_x[16] = 16.682320 <-> 3F2EAC80

block_x[17] = 17.682320 <-> 3F2EAC80

Follow-on chaotic map

block_x[0] = 0.682320 <-> 0.860477

…

block_x[2] = 2.682321 <-> 0.000000

…

block_x[16] = 16.682320 <-> 0.671719

block_x[17] = 17.682320 <-> 0.671719

Follow-on linear map

block_x[0] = 0.682320 <-> 0.000016

…

block_x[17]=17.682320 <-> 0.000016

The chaotic detection map outputs X_E's (fractional part in the first column) are not the identical across the blocks, and the differences are significant enough to be noticed when printed under C float format. The differences are more clearly seen in hexadecimal format; here the block number has been subtracted from the first column number. The outputs of the follow-on chaotic-map X_F's more clearly show significant deviations as these small precision differences in X_E's are non-linearly amplified. As an additional step, we also computed the outputs of a follow-on linear-map, $M(X) = X + \delta$, which shows that these differences are inconsequential, and it also shows that some linear maps do not provide the needed detection capability.

The outputs from Quadro 600 GPU are shown below using the tent map, wherein the results are qualitatively similar to Titan K20X GPU but the details differ.

Chaotic detection map:

block_x[0] = 0.170387 <-> 3E2E79D8

…

block_x[2] = 2.170387 <-> 3E2E79E0

…

block_x[8] = 8.170386 <-> 3E2E79C0

…

block_x[16] = 16.170387 <-> 3E2E7A00

Follow-on chaotic map:

block_x[0] = 0.170387 <-> 0.313038

…

block_x[2] = 2.170387 <-> 0.793185

…

block_x[8] = 8.170386 <-> 0.459723

…

block_x[16] = 16.170387 <-> 0.903821

Follow-on linear map:

block_x[0] = 0.170387 <-> 0.000041

…

block_x[16] = 16.170387 <-> 0.000041

The transition points of X_E are different in this case compared to the logistic map case, and in both cases they varied based on the number of blocks used by the CUDA kernel. But, these outputs are the same across all four GPUs used in our tests. These artifacts are related to the real number precision on GPU blocks. Similar precision effects also occur in CPU cores, and the application codes account for them in some cases by using double precision computations. Except on K20X GPUs, only single precision is supported on GPUs used in our tests. But, even when the same single precision (C float) operations are used, these precision effects are different between CPU cores and GPU blocks. To compare to CPU tests, we added the core number to X_E and subtracted it on host core, and no differences were found in X_E's using C float print; the largest number of cores we tested is 48, and such precision effects may indeed manifest when larger numbers are added. Consequently, if not adequately accounted for, these precision differences could lead to potentially unpredictable results in certain non-linear computations, particularly if automated tools are used to convert CPU codes to CPU-GPU hybrid systems.

4.4. Hybrid systems diagnosis

For the purposes of this subsection hybrid systems are considered as consisting of a mixture of CPU and GPU cores. Results are presented for three such systems using OpenCL kernels to perform the chaotic-map detection. The simplest system is the platform on a Macbook Pro which OpenCL detects as a single OpenCL platform with three different compute devices:

Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz, an Intel HD Graphics 4000 device, and an NVIDIA GeForce GT 650 M. Below are representative partial outputs for the single and double precision FPU tent map computations. Not show are computations for the integer ALU and EMU. Diagnosis Summary:

Device 0, SP-ALU : 3F1B0A10, DP-ALU : 3FE9B1B7A9B71338

Device 1, SP-ALU : 3F1B0A10, DP-ALU : FF800000FF800000

Device 2, SP-ALU : 3F1B0A10, DP-ALU : 3FE9B1B7A9B71338

Note that the DP ALU results for device 1 (Intel HD Graphics 4000) differ from the other devices as device 1 does not possess double precision capability.

The second hybrid system we consider is a node that OpenCL detects as consisting of two platforms with one and two compute devices, respectively. Platform 0 is an NVIDIA platform with a NVIDIA Tesla K20c device. Platform 1 is an AMD platform with device 0 an AMD Tahiti device (Firepro 9000) and device 1 an Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz. Below are representative partial outputs for the double precision FPU tent map computations. Not shown are calculations for the SP ALU, integer ALU, and EMU. All systems return the same result in the absence of errors. Diagnosis Summary:

Platform 0, device 0, DP-ALU : 3FE9B1B7A9B71338

Platform 1, device 0, DP-ALU : 3FE9B1B7A9B71338

Platform 1, device 1, DP-ALU : 3FE9B1B7A9B71338

The third system considered is a node with an AMD A10-7850 K APU. OpenCL identifies it as one platform with two devices. Device 0 is identified as AMD Spectre which consists of 8 GCN cores and Device 1 is identified as AMD A10-7850 K APU which consists of 4 CPU cores. Below are representative partial outputs for the single and double precision FPU tent map computations. Not show are computations for the integer ALU and EMU.

Diagnosis summary:

Device 0, SP-ALU : 3F1B0A10, DP-ALU : 3FE9B1B7A9B71338

Device 1, SP-ALU : 3F1B0A10, DP-ALU : 3FE9B1B7A9B71338

4.5. Operational artifacts

Our diagnosis codes were originally developed for low-level hardware faults, such as in ALU and interconnects. During the tests, however, they detected certain artifacts, which could lead to inconsistencies and/or errors in some computations if not adequately accounted for:

a. *Tardy computations*: In some systems, GPUs are emulated on the nodes, particularly if they housed them previously, and the emulated codes run sequentially on CPUs and lead to tardy computations. Our codes explicitly check for physical GPUs, and detected such emulations on nodes. Also, in SN-MG tests, we explicitly scheduled kernels on devices with numbers outside the eight GPUs, and the computations on them were completed

sometimes with incorrect results. These tests call for suitable device checks by application codes.

b. *Data transfer errors*: When CUDA kernels are launched and the results are gathered using MPI, certain elements received zero values. The occurrence of these errors was random but the zero elements always appeared in the blocks whose number matched the node.

These results provide information of interest to systems operations and application development.

5. Conclusions

We described a method to quickly detect certain faults in hybrid computing systems consisting of multi-core processors and accelerators by utilizing chaotic map computations. Our implementation is based on pthreads for multi-core CPUs and MICs, and CUDA C and OpenCL kernels for GPUs. We presented experimental diagnosis results on five multi-core CPUs, one MIC, seven GPUs and three hybrid systems. Since the original systems are not faulty, we simulated certain faults in arithmetic operations, local and global memory elements, data paths, and processor interconnects, which were detected. In addition, these codes identified artifacts of non-uniform precisions of GPU blocks and tardy hybrid computations, which could be of interest to non-linear computations.

Deeper investigations are needed to characterize the class of faults detected by a given set of chaotic maps, augmentation and data movement operations. While the logistic and tent maps used in our tests was able to detect the simulated faults, it would be interesting to study different chaotic maps whose Lyapunov exponents closely match the specific faults to minimize the detection times. More generally, it would be interesting to study the class of diagnosis algorithms that are optimal for a given class of faults. In terms of implementations, it would be interesting to explore finer control of memory allocations and data paths in movement operations using NUMA to further refine the diagnoses. In addition, assignment operations under OpenACC, SHMEM and PGAS involve data movements across complex data paths, and it would be interesting to explore the faults that can be detected by using them for content-preserving data movement operations. The proposed chaotic maps can be embedded into applications to track their execution paths so that faults can be detected during their execution. More generally, the fault diagnosis codes could be an integral part of overall ecosystems needed for resilient computations, and it would be of interest to co-develop them.

Acknowledgements

This work is supported in part by the United States Department of Defense and used resources of the Computational Research and Development Programs, and is also supported in part by Applied Mathematics Program, Office of Advanced Computing Research, Department of

Energy at Oak Ridge National Laboratory managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725. We acknowledge the support of Oak Ridge Leadership Computing Facility (OLCF) in making Lens, Titan and Chester systems available, and the information and guidance from Don Maxwell of OLCF that contributed to implementation and testing.

Author details

Nageswara S. V. Rao[1]* and Bobby Philip[2]

*Address all correspondence to: raons@ornl.gov

1 Oak Ridge National Laboratory, Oak Ridge, TN, USA

2 Los Alamos National Laboratory, Los Alamos, NM, USA

References

[1] Vetter JS, editor. Contemporary High Performance Computing: From Petascale toward Exascale. Boca Raton, Florida, USA: Chapman and Hall/CRC Press; 2013

[2] Cappello F. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. Journal of High Performance Computing Applications. 2009; **23**(3):212-226

[3] Dongarra J, Beckman P, et al. The international exascale software roadmap. International Journal of High Performance Computer Applications. 2011;**25**(1):3-60

[4] Cappello F, Geist A, Gropp B, Kale S, Kramer B, Snir M. Towards exascale resilience. Journal of High Performance Computing Applications. 2011;**23**(4):374-388

[5] Li M, Ramachandran P, Sahoo SK, Adve SV, Adve VS, Zhou Y. Understanding the propagation of hard errors to software and implications for resilient system design. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2008

[6] Erez M, Jayasena N, Knight TJ, Dally WJ. Fault tolerance techniques for the merrimac streaming supercomputer. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC); 2005

[7] Li D, Chen Z, Wu P, Vetter JS. Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach. In: ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis; 2013

[8] Marotto FR. Snap-back repellers imply chaos in r^n. Journal of Mathematical Analysis and Applications. 1978;**63**:199-223

[9] NVIDIA. NVIDIA GeForce GTX 200 GPU Architectural Overview. Santa Clara, CA, USA: NVIDIA Corporation; 2008

[10] NVIDIA. NVIDIAs Next Generation CUDA Compute Architecture. Santa Clara, CA, USA: NVIDIA Corporation, Fermi; 2009

[11] NVIDIA. NVIDIAs Next Generation CUDA Compute Architecture. Kepler GK110/210. Santa Clara, CA, USA: NVIDIA Corporation; 2012

[12] NVIDIA. NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made. Santa Clara, CA, USA: NVIDIA Corporation; 2014

[13] Parker TS, Chua LO. Chaos: A tutorial for engineers. Proceedings of the IEEE. 1987;**75**(8): 982-1008

[14] Rahman R. Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers. 1st ed. Berkely, CA, USA: Apress; 2013

[15] Rao NSV. Chaotic-identity maps for robustness estimation of exascale computations. In: 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS 2012); 2012

[16] Rao NSV. Fault detection in multi-core processors using chaotic maps. In: 3rd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS 2013); 2013

[17] Rao NSV. On undecidability aspects of resilient computations and implications to exascale. In: Resilience 2014: Seventh Workshop on Resiliency in High Performance Computing with Clouds, Grids, and Clusters; 2014

[18] Sahoo SK, Li M-L, Ramachandran P, Adve SV, Adve VS, Zhou Y. Using likely program invariants to detect hardware errors. In: International Conference on Dependable Systems and Networks; 2008

[19] HPL—A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. Available from: http://www.netlib.org/benchmark/hpl

[20] Wiggins S. Introduction to Applied Nonlinear Dynamical Systems and Chaos. New York: Springer-Verlag; 1990

[21] Alligood KT, Sauer TD, York JA. Chaos: An Introduction to Dynamical Systems. New York: Springer-Verlag; 1997

[22] Fujiwara H, Toida S. The complexity of fault detection problems for combinational logic circuits. IEEE Transactions on Computers. 1982;**C-31**(6):553-560

[23] Chen Z. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming; 2013

[24] Davies T, Chen X. Correcting soft errors online in lu factorization. In: Symposium on High-Performance Parallel and Distributed Computing; 2013

[25] Huang Y, Kintala C. Software fault tolerance of the application layer. In: Lyu MR, editor. Software Fault Tolerance. New York, NY, USA: John Wiley & Sons, Inc; 1995. pp. 231-248

[26] Jia Y, Luszczek P, Bosilca G, Dongarra J. CPU-GPU hybrid bidiagonal reduction with soft error resilience. In: Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA); 2013

[27] Carbin M, Misailovic S, Rinard MC. Verifying quantitaive realiability for programs that execute on unreliable hardware. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA); 2013

[28] de Kruijif M, Nomura S, Sankaralingam K. Relax: An architectural framework for software recovery of hardware faults. In: International Symposium on Computer Architecture (ISCA); 2010

[29] Gao J, Cao Y, Tung WW, Hu J. Multiscale Analysis of Complex Time Series. Hoboken, NJ, USA: John Wiley and Sons; 2007

[30] Hilborn RC. Chaos and Nonlinear Dynamics. Oxford, England, UK: Oxford University Press; 1994

[31] Intel. Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual. Santa Clara, CA, USA: Intel Corporation; 2012

[32] Intel. Intel Xeon Phi X100 Family Coprocessor—The Architecture. Santa Clara, CA, USA: Intel Corporation; 2012

[33] Jeffers J, Reinders J. Intel Xeon Phi Coprocessor Architecture High Performance Programming. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2013

[34] Kirk DB, Hwu WW. Progammng Massively Prallel Processors: A Hands-on Approach. Second ed. Morgan Kaufman Pub; 2013

[35] AMD. AMD Graphics Core Next (GCN) Architecture. Santa Clara, CA, USA: Advanced Micro Devices, Inc; 2012

IntechOpen